Best Case Analysis

- Least amount of (time) resource ever needed by algorithm
- Achieved when incoming list is \textit{already sorted} in increasing order
- Inner loop is never iterated
- Cost is given by:

\[
T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_5 (n-1) \\
= (c_1 + c_2 + c_3 + c_4 + c_5)n - (c_2 + c_4 + c_5 + c_6) \\
= an + b
\]

- Linear function of \( n \)
Worst Case Analysis

- Greatest amount of (time) resource ever needed by algorithm
- Achieved when incoming list is in reverse order
- Inner loop is iterated the maximum number of times, i.e., \( t_j = j \)
- Therefore, the cost will be:

\[
T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 - 1) + c_6(n(n-1)/2) \\
+ c_7(n(n-1)/2) + c_8(n-1) \\
= (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1+c_2+c_4+c_5/2 - c_6/2 - c_7/2 + c_8)n \\
- (c_2 + c_4 + c_5 + c_8)
\]

- Quadratic function of \( n \)

Future Analyses

- For the most part, subsequent analyses will focus on:
  - Worst-case running time
    - Upper bound on running time for any input
  - Average-case analysis
    - Expected running time over all inputs
- Often, worst-case and average-case have the same “order of growth”
Order of Growth

- Simplifying abstraction: interested in rate of growth or order of growth of the running time of the algorithm
- Allows us to compare algorithms without worrying about implementation performance
- Usually only highest order term without constant coefficient is taken
- Uses “theta” notation
  - Best case of insertion sort is $\Theta(n)$
  - Worst case of insertion sort is $\Theta(n^2)$

Designing Algorithms

- Several techniques/patterns for designing algorithms exist
- Incremental approach: builds the solution one component at a time
- Divide-and-conquer approach: breaks original problem into several smaller instances of the same problem
  - Results in recursive algorithms
  - Easy to analyze complexity using proven techniques
Divide-and-Conquer

- Technique (or paradigm) involves:
  - “Divide” stage: Express problem in terms of several smaller subproblems
  - “Conquer” stage: Solve the smaller subproblems by applying solution recursively – smallest subproblems may be solved directly
  - “Combine” stage: Construct the solution to original problem from solutions of smaller subproblem

Merge Sort Strategy

- **Divide stage**: Split the \( n \)-element sequence into two subsequences of \( n/2 \) elements each
- **Conquer stage**: Recursively sort the two subsequences
- **Combine stage**: Merge the two sorted subsequences into one sorted sequence (the solution)
Merging Sorted Sequences

- Combines the sorted subarrays $A[p..q]$ and $A[q+1..r]$ into one sorted array $A[p..r]$
- Makes use of two working arrays $L$ and $R$ which initially hold copies of the two subarrays
- Makes use of sentinel value ($\infty$) as last element to simplify logic
Merge Sort Algorithm

\begin{algorithm}
\textbf{MERGE-SORT}(A, p, r)
\begin{algorithmic}[1]
\STATE\textbf{if} $p < r$
\STATE\quad $q \leftarrow \lceil (p + r) / 2 \rceil$
\STATE\quad $T(n/2)$
\STATE\quad $T(n/2)$
\STATE\quad $T(n)$
\STATE\quad \textbf{MERGE-SORT}(A, p, q)
\STATE\quad \textbf{MERGE-SORT}(A, q + 1, r)
\STATE\quad \textbf{MERGE}(A, p, q, r)
\end{algorithmic}
\end{algorithm}

$T(n) = 2T(n/2) + \Theta(n)$

Analysis of Merge Sort

Analysis of recursive calls ...
Analysis of Merge Sort

\[ T(n) = cn(\lg n + 1) \]
\[ = cn\lg n + cn \]

\[ T(n) \text{ is } \Theta(n \lg n) \]